Debug Tutorial

Example Session

Instructions Session Output File: Makefile File: galaxy.h File: collapse.c File: create.c File: form.c File: main.c

Debug Tutorial: Introduction

Debug is an interactive command line debugger for applications running on the Teraflops system.

- Supports TOS and Cougar applications (core is coming)
- Supports F77, C, C++ languages, including combinations thereof (F90 is coming)
- dbx-like command set + parallel extensions (e.g. context, msgq, MPI)
- Scalable set of processes + data reduction assist user when viewing large amounts of output
- Symbolic debug support for programs compiled with -g

Debug Tutorial: Current Command Set Summary

The current set of commands available under debug are summarized by the **help** command as follows:

```
Execution and Tracing Commands
  run        Begin executing the program being debugged.
  wait       Wait for processes to stop running.
  cont       Continue execution from where it stopped.
  trace      Print a message before a procedure or source line is executed.
  stop       Stop execution when a program location is executed or a memory location
is accessed.
  halt       Stop program execution immediately.
  status     List all breakpoints and tracepoints currently set.
  delete     Remove breakpoints and tracepoints.
  step       Execute the next source line(s), stepping into functions.
  next       Execute the next source line(s), stepping over functions.

Naming, Printing and Displaying Data
  print      Print the value of an expression, an address, or register(s).
  whatis     Print the type of the given identifier.
  assign     Assign a value to a variable, address, or register.
  set        Assign a value to a variable, address, or register.
  where      List all, or the top n, active functions on the stack.

File Access Commands
  list       List the current or specified source line(s) or procedure.
  use        Print, set, add, or remove search path directories.

Commands for Parallel Processing
  commshow   Display MPI communicator or handles.
  context    Set or display the debug context.
  sendqueue  Display messages sent but not yet received.
  recvqueue  Display message receive requests posted but not satisfied.
  process    Display state information about user processes controlled by debug.

Miscellaneous Commands
  <Ctrl-C>   Interrupt the current command and give a new prompt.
  alias      Create an alias or display aliases.
  unalias    Delete previously-defined aliases.
  ?
  help       Display a synopsis of debug commands, or a help message.
  setv       Set or display the value of a variable, or display all variables.
  unsetv     Delete previously-defined debugger variables.
  source     Read and execute debug commands from a file.
  exit
  quit       Terminate the debug session and exit debug.
  debug      Load specified program for debugging.
  kill       Terminate and remove processes.
```

```
Machine-Level Commands
  tracei     Print a message just before an address is executed.
  stopi      Stop execution just before an address is executed.
  stepi      Single step machine instructions, stepping into functions.
  nexti      Single step machine instructions, stepping over functions.
  listi      Display machine code listing.
```

Some general rules which apply across the command set are as follows:

- Variable names are evaluated using scope of current point of execution unless specified explicitly. The syntax for scope specification is

  [`*sourcefile*`][*procedure*` | *linenumber*`] *variable*

  NOTE: The ` above is a back-quote character.
- File names may not include shell metacharacters except for ~.
- Rules for constructing a valid expression follow those of the target language except that procedure calls, assignment operators and type casts are not allowed.

Debug Tutorial: How to Load an Executable

Invocation syntax:

debug [**-n**] [**-s** *startup*] [*yod_args*] [*prog_name*] [*prog_args*]

Command syntax:

debug [**-n**] [*yod_args*] [*prog_name*] [*prog_args*] [**<** *input_file*]
[**>** *output_file*]

The command line debugger runs native on the Teraflops system. It is invoked by simply entering **debug**.

- The program name can be included on the invocation command line resulting in its being loaded prior to the first debug prompt.
- Alternatively, the **debug** command can be used to (re)load the program after invocation.
- A TOS application load requires the use of the **-n** switch.
- A Cougar application load is the same as a *yod* command line, simply replace *yod* with the **debug** command.
- Startup files specified at invocation are read immediately followed by *.debuginit* (located in cwd or $HOME). These are useful for defining personal command aliases and

speeding up the setup process for repetitive debug sessions.

- Source location search paths are specified with the **use** command.
- **debug** without arguments gives information about a load program and general debug environment.

Examples:

In this example, a parallel Cougar application is loaded after the debugger was invoked, using the **debug** command.

```
Debug  debug -sz 16 hello 1 2 3 abc  outputlog
 *** reading symbol table for /home/karla/hello...

 *** initializing Debug for parallel application...
 *** load complete
(all)  debug
Debugger Status:
  Mode      : parallel
  Program   : /home/karla/hello
  Arguments : 1 2 3 abc
  Input     :
  Output    : outputlog
  Yod       : /cougar/bin/yod
  YodArgs   : -sz 16
  LogFile   :
  More      : ON
  MsgStyle  : NX
```

This example loads a TOS application on the invocation line of the debugger.

```
{jethro:70} debug -n hello.osf

 *** Debug (Parallel Debugger), Release 1.6 beta
 *** Copyright (c) 1990,1991,1992,1993,1994,1995,1996 Intel Corporation

 *** reading symbol table for /home/karla/hello.osf...

 *** load complete
(host)  debug
```

```
Debugger Status:
  Mode      : non-parallel
  Program   : /home/karla/hello.osf
  Arguments :
  Input     :
  Output    :
  Yod       : /cougar/bin/yod
  YodArgs   :
  LogFile   :
  More      : ON
  MsgStyle  : NX
```

Debug Tutorial: How to Set Context

Command syntax:

context ({ **all** | *node_list* } **)**
context (*comm_handle* **:** { **all** | *rank_list* } **)**
commshow [*context*] [*expression* | *data_address*]

A context defines the set of processes to which a command applies.

- Prompt shows default context. It initially contains all nodes on which a program is loaded.
- **context** command changes the default context.
- Alternatively, a context can be specified for an individual command which overrides the default.
- A TOS application is given a context of *host*.
- A Cougar application context consists of one or more logical node numbers, *all*, or a communicator handle and rank list.
- **context** without arguments prints entire node list.
- **commshow** without arguments prints list of communicator handles.
- **commshow** given a MPI_Comm type variable prints its

communicator handle.

Example context specifications:

context (host)
context (0)
context (all)
context (0,1)
context (1..20, 25, 30..35)
context (COMMWORLD:all)
context (COMM1:0,5)
context (COMMSELF4:0)
context (ICOMM2:all)

Example communicator handles as displayed by **commshow** :

In the case below where **commshow** is given a specific variable to find the handle of, it is normal for an error to be reported for any process which is not contained in that communicator.

```
(COMMWORLD:all)  commshow
Intracommunicators:
Name        Size    Rank (in COMMWORLD)
=======     =====   ===================
COMMWORLD      6    0..5
COMM1          5    1..5

COMMSELF[0..5]

Intercommunicators:
Name        Intracommunicator Pair
=======     =====================

(COMMWORLD:all)  commshow other
  ***** (COMMWORLD:0) *****

 ** comm3.c{}main(int, char**)#34 other **
```

```
ERROR: cannot get communicator information
***    Null pointer argument

  ***** (COMMWORLD:1..5) *****

 ** comm3.c{}main(int, char**)#34 other **
other = COMM1
```

Intel Tutorial Notes

Previous -- Next

Debug Tutorial: How to View Process State

Command syntax:

process [*context*] [**change**] [**full**]

Process states fall into two general catergories, running and stopped. Many commands cannot act upon a process in a running state and will print an error if this is attempted. The **halt** command can be used to put a process into a stopped state so that it can be examined.

- **process** lists the state of all processes in the context. Any process in a stopped state includes its current location and the reason it stopped. An '*' to the left of the context column indicates a state change since the last time that process's state was displayed. (The '' currently has no meaning.)
- A program which has been loaded is automatically executed to the first line of user code and placed in the *Initial* state.
- A program is automatically stopped just prior to executing the exit procedure and placed in the *Exiting* state. Any process continued past this point enters the *Exited* state and is no longer valid.
- Running states consist of *Executing* and *Stepping*. A

program executing a blocking receive will be in one of these states.
- The **halt** command stops execution and places the process in the *Interrupted* state.
- A process which just completed a **next, step, nexti, or stepi** command will be in the *Stepped* state.
- A process which encountered a code or data breakpoint will be in the *Breakpoint* state. The number of the breakpoint encountered is included in the reason field.
- A process for which a signal has arrived is placed in the *Signaled* state. The name of the signal is included in the reason field. The signal handler (default or user-defined) will not be executed until execution is resumed.
- The **change** switch causes only processes' with a state change to be displayed. The **full** switch causes a procedure name to be fully qualified.
- The **step**, **next**, and **wait** commands automatically display process state when they complete.

Example process state display:

```
       Context              State      Reason    Location      Procedure
  ===================== ============ ========= ==========
====================
 (0)                    Initial                Line 16    main()
 (1,3)                  Stepped                0x0003b410 _doprnt()
*(2)                    Stepped                Line 19    main()
 (4)                    Breakpoint   C Bp2     Line 20    main()
*(5)                    Signaled     SIGSEGV   Line 13    one()
```

Debug Tutorial: How to View Source Code

Command syntax:

list [*context*] [*start_line* | *procedure*] [*count*]

Source files are automatically searched for in the current directory. The **use** command should be used to add other directories to be searched or to change the order in which the paths are to be searched.

- The **list** command without arguments displays 10 source lines starting at the current point of execution. **count** can be specified to change the default count and it will carry to subsequent **list** commands.
- If a procedure name is specified, a 'window' of count lines before and after the procedure's entry point is listed.
- Subsequent uses of **list** continue the listing of source lines from where the prior **list** left off if no arguments are specified. Executing a process causes the starting point of the next **list** to be reset.
- **list** may be used when processes are not stopped if a line number or procedure name is specified.
- The source listing is numbered with a '*' preceding a line number to indicate a breakpoint can be set there.

- **listi** displays disassembled code, intermixed with source line numbers if the program was compiled with *-g*.

Examples:

In this example, **list** was given no arguments so the source listing will begin at the current point of execution for each process. Process 0 is stopped at line 7 while the other processes are at line 16.

```
(all)  list
***** (0) *****
./hello.c
* 7    int a = 0;
  8    int *ptr;
  9
* 10    i++;
* 11    b=4;
* 12    i = a+b;
* 13 }
  14
  15 main()
* 16 {
  17    int i;
***** (1..3) *****
./hello.c
* 16 {
  17    int i;
* 18    int p1 = 555;
* 19    int p2 = 1234;
  20
* 21    printf("Hello has started on %d\n", mynode());
* 22    one( p1 );
* 23    printf("mystring=\"%s\"\n", mystring);
* 24 }
```

In this example, a disassembly is requested for procedure *one*.

```
(all)  listi one
  ***** (all) *****
  hello.c{}one(int)#5
00020130: 83ec0c                sub      0xc,esp
00020133: 896c2408              mov      ebp,8(esp)
00020137: 8d6c2408              lea      8(esp),ebp
  hello.c{}one(int)#7
0002013b: c745fc00000000        mov      0x0,-4(ebp)
  hello.c{}one(int)#10
```

```
00020142: 8b4508                mov      8(ebp),eax
00020145: 40                    inc      eax
00020146: 894508                mov      eax,8(ebp)
  hello.c{}one(int)#11
00020149: c745f804000000        mov      0x4,-8(ebp)
  hello.c{}one(int)#12
00020150: 8b45f8                mov      -8(ebp),eax
00020153: 0345fc                add      -4(ebp),eax
00020156: 894508                mov      eax,8(ebp)
  hello.c{}one(int)#13
00020159: 89ec                  mov      ebp,esp
0002015b: 5d                    pop      ebp
0002015c: c3                    ret
```

Intel Tutorial Notes

Previous -- Next

Debug Tutorial: How to Set a Breakpoint

Command syntax:

stop [*context*] [**in** | **at**] { *line_num* | *procedure* } [**,***count* | **if** *condition*]
stop [*context*] { **rw** | **w** } { *expression* | *data_address* } [**,***count* | **if** *condition*]
stopi [*context*] [**at**] *text_address* [**,***count* | **if** *condition*]
trace [*context*] [**in** | **at**] { *line_num* | *procedure* } [**,***count* | **if** *condition*]
tracei [*context*] [**at**] *text_address* [**,***count* | **if** *condition*]
delete [*context*] { **all** | *bkpt_num* }
status [*context*] [**>***filename*]

Breakpoints force execution of a program to stop at points of interest to allow examination of data, registers, stack, and message queues.

A code breakpoint is placed on an instruction address. Execution is stopped *before* that instruction is executed. A data breakpoint, which is referred to here as a watchpoint, is placed on a data address. In this case, execution stops *after* that address has been accessed.

- **stop** followed by a line number sets a code breakpoint at the start of that source line. ***Note: Do not set a breakpoint on a loop statement and expect it to be hit***

while the loop is executed. Specify the first line within the loop instead.

- **stop** followed by a procedure name sets a code breakpoint after the preamble of that procedure. Procedure parameters are therefore defined when execution stops. An attempt to set a breakpoint at the line number that corresponds to this location will fail.
- **stop** with a **-rw** or **-w** switch sets a watchpoint on the variable or address specified. The size of a watchpoint object is currently always assumed to be 4 bytes. ***Note: The process command's location field indicates the next instruction to be executed. Thus, watchpoints which fire at the very end of a source line will appear with the line number of the next source line to be executed rather than the source line on which the watchpoint occurred.***
- Watchpoints are set using hardware registers which makes them fast, but limits their number to 4.
- **,***count* or **if** *condition* can be specified to delay the reported occurrence of a breakpoint or watchpoint until the condition is met. The count option results in the breakpoint or watchpoint being reported after every *count* times it is encountered. A *condition* is a simple expression that evaluates to True (non-zero) or False (0). The breakpoint or watchpoint is only reported when the condition evaluates to True. A *condition* consists of equivalence operators and logical operators. The syntax used must be the same as that of the program under debug, e.g. > for C and *.GT.* for Fortran.

- **stopi** sets a code breakpoint on an instruction address.
- **trace** and **tracei** are similar to code breakpoints. When they are encountered a message is printed and execution continues.
- **delete** removes breakpoints, watchpoints, and tracepoints.
- **status** lists breakpoints, watchpoints, and tracepoints along with their associated number. If the redirection option is specified () a file containing these breakpoints, watchpoints, and tracepoints is created. They are written to the file in the form of debug commands suitable for use as a startup file or **source** command input file.

Examples:

```
(0)  stop 10
(0)  stop one
(0)  stop 25,2
(0)  stop -rw myvar if myvar < 0
(0)  stop -w one`i
(0)  status

( 1) stop at line 10:hello.c:one():(0)
( 2) stop in one():hello.c:one():(0)
( 3) stop at line 25:hello.c:main():(0)
( 4) stop if access myvar:::(0)
( 5) stop if write i:hello.c:one():(0)
(0)  delete all
```

Debug Tutorial: How to Control Execution

Command syntax:

cont [*context*]
run [*prog_args*] [**<** *input_file*] [**>** *output_file*]
wait [*context*]
next [*context*] [*count*]
nexti [*context*] [*count*]
step [*context*] [*count*]
stepi [*context*] [*count*]

Program execution is controlled by first setting breakpoints and/or watchpoints and then running the application until one of them is encountered. Alternatively, execution can be stepped along one source line (or one instruction) at a time.

- **cont** command resumes execution from the current location.
- **run** command (re)starts execution at the beginning of the program. Application arguments and I/O redirection of the preceding load or run command are reused unless specified. Breakpoints, watchpoints and tracepoints are preserved.
- **wait** must be used with **cont** and **run** to allow terminal I/O to occur while the program is executing. The debugger prompt will not appear until all processes in the context have reached a stopped state. During this time, any keyboard input will be consumed by the application and output from the application is printed to the screen immediately.

- If **wait** is not used, application output is only printed to the screen between execution of debugger commands. In this case, hitting the Return key several times will cause the buffered application output to be displayed to the screen.
- Use <Ctrl-C> to interrupt a debug command such as **wait**. ***Note: <Ctrl-C> DOES NOT STOP PROGRAM EXECUTION!!***
- Use the **halt** command to interrupt program execution. ***Note: halt DOES STOP PROGRAM EXECUTION!!***
- **next** causes a single source line to be executed. A procedure call will be treated as a single source line, thus stepping over any procedures.
- **step** acts the same as **next** except that procedures are stepped into and executed line by line. A procedure which does not contain line number information is treated as a single source line and stepped over.
- **stepi** and **nexti** cause a single instruction to be executed.
- Use *count* with any of the **next** and **step** commands to execute multiple lines.

Examples:

```
(all)  cont;wait
Hello has started on 0
Hello has started on 1
Hello has started on 2
Hello has started on 3
Hello has started on 4
Hello has started on 5
        Context              State      Reason    Location     Procedure
  ====================== ============ ========= ==========
====================
*(all)                  Breakpoint   C Bp1     Line 22    main()
(all)  step(0)
        Context              State      Reason    Location     Procedure
  ====================== ============ ========= ==========
====================
*(0)                    Stepped                Line 5     one()
(all)  next(2,3)
        Context              State      Reason    Location     Procedure
  ====================== ============ ========= ==========
====================
*(2,3)                  Stepped                Line 24    main()
(all)  run;wait
/cougar/bin/yod: Received SIGINT (2)
 *** initializing Debug for parallel application...
Hello has started on 0
Hello has started on 1
Hello has started on 2
Hello has started on 3
Hello has started on 4
Hello has started on 5
        Context              State      Reason    Location     Procedure
  ====================== ============ ========= ==========
====================
*(all)                  Breakpoint   C Bp1     Line 22    main()
```

Debug Tutorial: How to Examine Stack

Command syntax:

where [*context*] [*count*]

A stack traceback lists the call sequence which got the program to its current execution point. The traceback is displayed such that the first procedure listed indicates the current point of execution.

- **where** displays a stack traceback. If *count* is specified, the stack display is limited to the top *count* procedures.
- Procedure names are displayed even when the program was not compiled with -g. If a function name cannot be determined, "????()" is displayed in its place. This is currently seen at the bottom of all stack tracebacks and can be ignored.

Examples:

```
(all)  where
***** (0) *****
one(int) [hello.c{} #7]
main(void) [hello.c{} #24]
cstart() [unknown{} 0x00025646]
????() [hello.c{} 0x00020120]
***** (1) *****
main(void) [hello.c{} #24]
cstart() [unknown{} 0x00025646]
????() [hello.c{} 0x00020120]
(all)  where(0) 1
***** (0) *****
one(int) [hello.c{} #7]
```

Debug Tutorial: How to Examine Message Queues

Command syntax:

sendqueue [*context*] [**all**]
recvqueue [*context*] [**all**]

Programs doing message passing may have unexpected results or hang because message sends and receives either match unexpectedly or not at all. The ability to view messages and receives sitting in system queues can provide critical information to resolve this kind of programming error.

- **sendqueue** displays all messages sent to, but not received by, the processes in the context.
- **recvqueue** displays all unsatisfied receives posted by the processes in the context
- When MPI communicators are used for message passing, the messages displayed are filtered so that only those messages sent within the communicator specified in the context are displayed. Use the **all** switch to eliminate this additional filter.
- If a posted receive specified a wild card source process or tag, it appears as a -1 for NX messages and *ANY* for MPI

messages.

- If a communicator no longer exists for a message in one of these queues it is displayed with a communicator handle of COMMUNKNOWN and is only seen when the **all** switch is used.
- Message length is always in bytes, not number of elements.
- Only point-to-point messages are currently included in these displays.

Examples:

```
(COMMWORLD:all)  sendqueue

 *** Unreceived messages in (COMMWORLD:all)

                                                        Msg Length
      Source           Destination        Msg Tag      (in bytes)
================== ================== ============ ============
(COMMWORLD:0)       (COMMWORLD:1)       22           10
(COMMWORLD:0)       (COMMWORLD:2)       22           10
(COMMWORLD:0)       (COMMWORLD:3)       22           10
(COMMWORLD:0)       (COMMWORLD:4)       22           10
(COMMWORLD:0)       (COMMWORLD:5)       22           10
(COMMWORLD:all)  sendqueue -all

 *** Unreceived messages in (COMMWORLD:all)

                                                        Msg Length
      Source           Destination        Msg Tag      (in bytes)
================== ================== ============ ============
(COMMWORLD:0)       (COMMWORLD:1)       22           10
(COMM1:0)           (COMM1:0)           33           10
(COMMWORLD:0)       (COMMWORLD:2)       22           10
(COMM1:0)           (COMM1:1)           33           10
(COMMWORLD:0)       (COMMWORLD:3)       22           10
(COMM1:0)           (COMM1:2)           33           10
(COMMWORLD:0)       (COMMWORLD:4)       22           10
(COMM1:0)           (COMM1:3)           33           10
(COMMWORLD:0)       (COMMWORLD:5)       22           10
(COMM1:0)           (COMM1:4)           33           10
```

```
 *** Unreceived messages in (all)

                                                      Msg Length
      Source          Destination       Msg Type     (in bytes)
================== ================== ============ ============
(COMMWORLD:all)  sendqueue(comm1:all)

 *** Unreceived messages in (COMM1:all)

                                                      Msg Length
      Source          Destination       Msg Tag      (in bytes)
================== ================== ============ ============
(COMM1:0)          (COMM1:0)          33           10
(COMM1:0)          (COMM1:1)          33           10
(COMM1:0)          (COMM1:2)          33           10
(COMM1:0)          (COMM1:3)          33           10
(COMM1:0)          (COMM1:4)          33           10
(COMMWORLD:all)  recvqueue

 *** Unsatisfied receives posted in (COMMWORLD:all)

                                                      Msg Length
  Recv Posted By      For Msg From      Msg Tag      (in bytes)
================== ================== ============ ============
(COMMWORLD:all)  recvqueue -all

 *** Unsatisfied receives posted in (COMMWORLD:all)

                                                      Msg Length
  Recv Posted By      For Msg From      Msg Tag      (in bytes)
================== ================== ============ ============
(COMM1:0)          (COMM1:0)          22           10
(COMM1:1)          (COMM1:0)          22           10
(COMM1:2)          (COMM1:0)          22           10
(COMM1:3)          (COMM1:0)          22           10
(COMM1:4)          (COMM1:0)          22           10

 *** Unsatisfied receives posted in (all)

                                                      Msg Length
  Recv Posted By      For Msg From      Msg Type     (in bytes)
================== ================== ============ ============
(COMMWORLD:all)  recvqueue(comm1:0)

 *** Unsatisfied receives posted in (COMM1:0)

                                                      Msg Length
  Recv Posted By      For Msg From      Msg Tag      (in bytes)
================== ================== ============ ============
(COMM1:0)          (COMM1:0)          22           10
```

Debug Tutorial: How to Examine Data

Command syntax:

print [*context*] [*format*] { *expression* | *data_address* } [**,***count*]
print [*context*] [*format*] **-***register_name*
print [*context*] **-reg**
whatis [*context*] *identifier*

The debugger provides for the examination of data items in the program without having to insert print statements.

- **print** displays the value of a symbol or an expression.
- **print** with a structure name displays the value of each element in the structure.
- **print** with an array name displays the entire array from the beginning. Use **,***count* to control the number of elements printed.
- **print** with **-reg** switch or a specific register name switch displays the contents of the register(s).
- **print** with a simple, unparenthesized number assumes the number to be a data address and attempts to print the value stored at that address. *Note: This is an anomaly which will be changed when the **address** command is*

functional.

- Format switches can be used to convert the form in which a value is displayed (available switches: **d, o, x, l, s, a, f, F, m, M**).
- **whatis** displays the data type for a given identifier.
- To get the data address of a variable, use '&' in C and 'LOC()' in Fortran.

Examples:

```
(all)  print l
***** (0) *****
 ** `bob.c`level3`53`l **
l = 3
***** (1) *****
 ** `bob.c`level1`68`l **
l = 1
***** (2) *****
 ** `bob.c`level2`60`l **
l = 2
(all)  print main`S1
***** (all) *****
 ** `bob.c`main`75`S1 **
struct AnyStruct {
    int  i1 = 1
    int  i2 = 0
    char  c1 = '\0000'
    char  * cPtr1 = 0x00000000
}
(all)  print(0) carry
***** (0) *****
 ** `bob.c`level3`53`carry **
  carry[0] = 'a'
  carry[1] = 'b'
  carry[2] = 'c'
  carry[3] = 'd'
  carry[4] = 'e'
(all)  print globalCharPtr
***** (all) *****
 ** `bob.c`main`101`globalCharPtr **
globalCharPtr = 0x008200e0
(all)  print *globalCharPtr
***** (all) *****
 ** `bob.c`main`101`*globalCharPtr **
```

```
*globalCharPtr = 'a'
(all)  print &ll
***** (all) *****
 ** `bob.c`main`101`&ll **
&ll = 0x7fc40434
(all)  whatis arry
***** (all) *****
int  arry[5][4]
(all)  whatis carry[0]
***** (all) *****
char
(all)  whatis S1
***** (all) *****
struct AnyStruct {
    int  i1;
    int  i2;
    char  c1;
    char  * cPtr1;
}
(all)  whatis level1i
***** (all) *****
int  level1(long )
```

Debug Tutorial: How to Modify Data

Command syntax:

set [*context*] *variable*[**,***count*] = *expression*
set [*context*] [*size_switch*] *address*[**,***count*] = *expression*
set [*context*] [*size_switch*] **-***register_name* = *expression*

It may be useful to modify a data item during runtime and continue execution to see what happens, thus avoiding a re-compilation.

- **set** modifies the contents of a specified variable, address, or register.
- **assign** is another name for this command.
- If an array name is specified, each element in the array will be set to the specified value.
- Assignment to an entire structure is not allowed. Must specify an individual element.
- Switches can be used to explicitly indicate the number of bytes to be modified when specifying an address or register (available switches: **b, s, l, d**).
- Modification of data items does not persist when re-running the program from the beginning.

Examples:

```
(all)  assign i = 5
(all)  assign carry[0]='A'
(all)  assign(2..3) arry,4=100
(all)  assign f1=3.3333
```

Debug Tutorial: How to Terminate Debug Session

Command syntax:

quit
exit

quit or **exit** will cause the loaded program to be terminated immediately and the debugger to exit.

Example:

```
(all)  quit
*** Debug exiting
/cougar/bin/yod: Received SIGINT (2)
```

Debug Tutorial: Instructions for Example

This example session will introduce you to basic *debug* commands.

You must first collect the example source files and Makefile and build the executable `galaxyf`. This example code is a small demonstration program that does NX message passing.

Starting debug

- Change your directory so that you are present in the directory where the example executable is located.
- Start debug on eight nodes:
  `debug -sz 8 galaxyf`
- debug should come up, pause while loading the processes, and then display the debug prompt containing the default context.
- If the source files for the example are in a different directory, add that directory to the source search path list with the use command:
  `use + galaxy`

Setting Context and Viewing Process State

- The context defines the set of processes to which a command is applied. The default context displayed in the prompt can be changed by issuing the context command with a new context argument. Change the context to include only 1 process and then change it back again:
  ```
  context(1)
  context(all)
  ```
- Most commands allow a context specification which causes the context to change only for the single command.
- The state of the processes can be determined using the process command. Display the state of the processes immediately after loading:
  `process`

Viewing Source and Setting Breakpoints

- A line numbered source code listing is given by the list command. You can see the line you are currently stopped at by entering:
  `list`
- Entering **list** again will continue listing from where the previous list left off:
  `list`
- Now list the lines where we will be setting some breakpoints:
  `list 40`
- Set breakpoints on lines 44, 46, and 48 using the stop command as follows:
  `stop 44; stop 46; stop 48`
- Use the status command to confirm the breakpoints are set:
  `status`
- Now execute the program with the cont command as follows:
  `cont;wait`
- Note the use of the **wait** command in conjunction with **cont** which allows any program terminal I/O to occur.
- Shortly, a process state display will appear (caused by the **wait**) which shows processes stopped at lines 44, 46, and 48.
- The context indicates which processes' are stopped at each breakpoint.

Viewing Data

- The contents of a variable is displayed using the print command. Print the value of a variable:
  `print my_node`
- The data values are unique for each process, so each is listed separately.
- Print the first 5 elements of the receive buffer array:
  `print recvLeft.fMessage,10`
- In this case the data values are the same across all of the nodes so a single list is displayed.

Forcing Message Blocking Situation

- Set a breakpoint on the message receive call in *form.c* procedure *Form()* at line 20:
  `` stop `form`20 ``
- Resume execution of the program:
  `cont;wait`

- After waiting awhile, you realize the program is taking longer than expected to reach the breakpoint. Interrupt the wait command:
  `<Ctrl-C>`
- Check the state of the processes:
  `process`
- Note that processes 2, 4, and 6 (for eight process load) have hit the breakpoint while the other processes remain in the *Executing* state.
- Stop these processes with the halt command:
  `halt`
- The previously *Executing* processes should now be in the *Interrupted* state as seen with the process command:
  `process`

Viewing the Stack

- The call stack is displayed with the where command:
  `where`
- You can now determine that the processes you halted are stopped within a routine that reflects a blocking receive was in progress.
- You can view the source code for a particular function of interest (e.g. to see the blocking call):
  `list Form`
- Note that a *window* of lines is displayed in this case.

Viewing the Message Queue

- Information about receives in progress is displayed with the recvqueue command:
  `recvqueue`
- Pending message information is viewed using the sendqueue command:
  `sendqueue`

Exiting Debug

- You can allow the program to finish executing at this point or simply exit the debugger. To finish the execution, remove all of the breakpoints using the delete command and continue the program:
  `delete all`
  `cont; wait`

To terminate the program and exit the debugger use the quit command (or **exit**):
`quit`

Debug Tutorial: Example Session Output

Go to:

- invoke debug
- use command
- context command
- process command
- list command
- stop command
- status command
- cont command
- print command
- interrupt wait command
- halt command
- where command
- recvqueue command
- sendqueue command
- delete command
- quit command

```
% debug -sz 8 galaxyf

 *** Debug (Parallel Debugger), Release 1.6 beta
 *** Copyright (c) 1990,1991,1992,1993,1994,1995,1996 Intel Corporation

 *** reading symbol table for /home/karla/galaxyf...

 *** initializing Debug for parallel application...
 *** load complete
```

```
(all) > use + galaxy

*** Global path list:
galaxy

(all) > context(1)

(1) > context(all)

(all)  process
       Context              State       Reason    Location     Procedure
  ====================== ============ ========= ========== ====================
>*(all)                  Initial                 Line 24    main()

(all) > list

***** (all) *****
galaxy/main.c
* 24     {
  25     int             num_nodes;
  26
  27
* 28     SetClock();
  29
* 30     my_node   = mynode();
* 31     num_nodes = numnodes();
  32
  33     /*\ Assert where messages will be sent to
  34     \*/
(all) > list
***** (all) *****
galaxy/main.c
  34     \*/
  35
* 36     DetermineRouting( my_node, num_nodes, &left_node, &right_node);
  37
* 38     InitializeMessages( my_node, num_nodes);
  39
  40     /*\ Distribute work
  41     \*/
  42
* 43     if ( my_node == 0)
* 44         Create();
(all) > list 40
***** (all) *****
galaxy/main.c
  40     /*\ Distribute work
  41     \*/
  42
* 43     if ( my_node == 0)
* 44         Create();

* 46         Collapse();
  47     else
* 48         Form();
  49
  50     /*\ Barrier
```

```
(all) > stop 44; stop 46; stop 48


(all) > status


( 1) stop at line 44:main.c:main():(all)
( 2) stop at line 46:main.c:main():(all)
( 3) stop at line 48:main.c:main():(all)

(all) > cont; wait

        Context             State      Reason    Location     Procedure
  ===================== ============ ========= ========= ===================
>*(0)                    Breakpoint   C Bp1     Line 44    main()
>*(1..6)                 Breakpoint   C Bp3     Line 48    main()
>*(7)                    Breakpoint   C Bp2     Line 46    main()

(all) > print my_node

***** (0) *****
 ** `main.c`main`44`my_node **
my_node = 0
***** (1) *****
 ** `main.c`main`44`my_node **
my_node = 1
***** (2) *****
 ** `main.c`main`44`my_node **
my_node = 2
***** (3) *****
 ** `main.c`main`44`my_node **
my_node = 3
***** (4) *****
 ** `main.c`main`44`my_node **
my_node = 4
***** (5) *****
 ** `main.c`main`44`my_node **
my_node = 5
***** (6) *****
 ** `main.c`main`44`my_node **
my_node = 6
***** (7) *****
 ** `main.c`main`44`my_node **
my_node = 7
(all) > print recvLeft.fMessage,10
***** (all) *****
 ** `main.c`main`44`recvLeft.fMessage **
  recvLeft.fMessage[0] = '\0000'
  recvLeft.fMessage[1] = '\0000'
  recvLeft.fMessage[2] = '\0000'
  recvLeft.fMessage[3] = '\0000'
  recvLeft.fMessage[4] = '\0000'
  recvLeft.fMessage[5] = '\0000'
  recvLeft.fMessage[6] = '\0000'
  recvLeft.fMessage[7] = '\0000'
  recvLeft.fMessage[8] = '\0000'
  recvLeft.fMessage[9] = '\0000'
(all) > stop `form`20
```

```
(all) > cont; wait

^C

(all) > process
        Context             State      Reason    Location     Procedure
  ===================== ============ ========= ========= ===================
>*(0,1,3,5,7)            Executing                          unknown fcn
>*(2,4,6)                Breakpoint   C Bp4     Line 20    Form()

(all) > halt

(all) > process
        Context             State      Reason    Location     Procedure
  ===================== ============ ========= ========= ===================
>*(0)                    Interrupted            0x00033697 def_vrecv()
>*(1,3,5,7)              Interrupted            0x0002c063 sptl_ind_msg_probe()
> (2,4,6)                Breakpoint   C Bp4     Line 20    Form()

(all) > where

***** (0) *****
def_vrecv() [unknown{} 0x00033697]
_reduce_short() [unknown{} 0x000309ba]
_gsync() [unknown{} 0x0002111b]
gsync() [unknown{} 0x00020fe0]
main(int, char**) [main.c{} #55]
cstart() [unknown{} 0x00028cf6]
????() [collapse.c{} 0x00020120]
***** (1) *****
sptl_ind_msg_probe() [unknown{} 0x0002c063]
_msgwait() [unknown{} 0x00025858]
_crecv() [unknown{} 0x000242fa]
crecv() [unknown{} 0x00024274]
Form(void) [form.c{} #16]
main(int, char**) [main.c{} #53]
cstart() [unknown{} 0x00028cf6]
????() [collapse.c{} 0x00020120]
***** (2,4,6) *****
Form(void) [form.c{} #20]
main(int, char**) [main.c{} #53]
cstart() [unknown{} 0x00028cf6]
????() [collapse.c{} 0x00020120]
***** (3,5) *****
sptl_ind_msg_probe() [unknown{} 0x0002c063]
_msgwait() [unknown{} 0x00025858]
_crecv() [unknown{} 0x000242fa]
crecv() [unknown{} 0x00024274]
Form(void) [form.c{} #16]
main(int, char**) [main.c{} #53]
cstart() [unknown{} 0x00028cf6]
????() [collapse.c{} 0x00020120]
***** (7) *****
sptl_ind_msg_probe() [unknown{} 0x0002c063]
_msgwait() [unknown{} 0x00025858]
_crecv() [unknown{} 0x000242fa]
crecv() [unknown{} 0x00024274]
Collapse(void) [collapse.c{} #16]
```

```
main(int, char**) [main.c{} #46]
cstart() [unknown{} 0x00028cf6]
????() [collapse.c{} 0x00020120]
(all) > list Form
***** (all) *****
galaxy/form.c
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <nx.h>
  4
  5 #include "galaxy.h"
  6
  7
  8
  9 void Form()
* 10     {
* 11     if ( my_node & 0x01)
  12         {
* 13         csend( 0, &sendLeft, sizeof( TMessage), left_node, 0);
* 14         csend( 0, &sendRight, sizeof( TMessage), right_node, 0);
* 15         crecv( 0, &recvLeft, sizeof( TMessage));
* 16         crecv( 0, &recvRight, sizeof( TMessage));
  17         }
  18     else
  19         {
* 20         crecv( 0, &recvLeft, sizeof( TMessage));

(all) > recvqueue

 *** Unsatisfied receives posted in (all)

                                                           Msg Length
  Recv Posted By      For Msg From      Msg Type    (in bytes)
================== ================== ============ ============
(1)                (-1)               0            68
(3)                (-1)               0            68
(5)                (-1)               0            68
(7)                (-1)               0            68

(all) > sendqueue

 *** Unreceived messages in (all)

                                                           Msg Length
      Source          Destination       Msg Type    (in bytes)
================== ================== ============ ============
(1)                (2)                0            68
(3)                (2)                0            68
(3)                (4)                0            68
(5)                (4)                0            68
(5)                (6)                0            68
(7)                (6)                0            68

(all) > delete all

(all) > cont; wait
End simulation
<1> -- <  0: Calculate from   0.00% to  12.50%>
```

```
<4> -- <  3: Calculate from  37.50% to  50.00%>
<2> -- <  1: Calculate from  12.50% to  25.00%>
<3> -- <  2: Calculate from  25.00% to  37.50%>
<6> -- <  5: Calculate from  62.50% to  75.00%>
<5> -- <  4: Calculate from  50.00% to  62.50%>
<7> -- <  0: Calculate from   0.00% to  12.50%>
...total time:  0.019430
<1> -- <  2: Calculate from  25.00% to  37.50%>
<4> -- <  5: Calculate from  62.50% to  75.00%>
<2> -- <  3: Calculate from  37.50% to  50.00%>
<3> -- <  4: Calculate from  50.00% to  62.50%>
<6> -- <  7: Calculate from  87.50% to 100.00%>
<5> -- <  6: Calculate from  75.00% to  87.50%>
<7> -- <  6: Calculate from  75.00% to  87.50%>
<0> -- <  1: Calculate from  12.50% to  25.00%>
<0> -- <  7: Calculate from  87.50% to 100.00%>
        Context             State      Reason    Location     Procedure
  ===================== ============ ========= ========= ===================
>*(all)                  Exiting                0x000404d0 exit()

(all) > quit

*** Debug exiting
/cougar/bin/yod: Received SIGINT (2)
```